ต้นฉบับไม่มีหน้า 354

No This Page 354 In Original

โปรเซสแต่ละอันที่เกิดขึ้นนี้ ปกติจะเป็นอิสระจากกัน โดยระบบปฏิบัติการและด้วยความร่วมมือของฮาร์ดแวร์นั้นจะป้องกันไม่ให้แต่ละโปรเซสล่วงล้ำเข้าไปแอบดูหรือแก้ไขอาณาบริเวณของโปรเซสอื่น ๆ ได้ ก็เพื่อความปลอดภัยด้วยอย่างหนึ่ง แต่ว่าระบบปฏิบัติการยูนิกซ์เปิดโอกาสให้แต่ละโปรเซสสามารถคุยกันได้คือส่งข้อมูลระหว่างกันได้ และระบบการคุยกันระหว่างโปรเซสนี้ (Inter Process Communication หรือ IPC) นับว่าเยี่ยมทีเดียว มีให้ผู้ใช้ได้เลือกหลายวิธี ตั้งแต่วิธีง่าย ๆ จนถึงวิธีที่ใช้ยากและมีประสิทธิภาพสูง ซึ่งบางวิธีก็สามารถขยายขอบเขตให้คุยกันข้ามเครื่องไปสู่โลกภายนอกได้เลย

กลับมาเรื่องของเชลล์ ผู้ที่ใช้ยูนิกซ์โดยเฉพาะเล่นกับเชลล์มาพอสมควร หลายคนจะประทับใจกับความสามารถของมัน ยูนิกซ์ทุกเครื่องจะมีโปรแกรมพื้นฐานที่อยู่ใน /bin, /usr /bin, /etc ฯลฯ มากมายที่เรานำมาใช้ให้เกิดประโยชน์ได้มากโดยอาศัยความยืดหยุ่นและความสามารถของเชลล์ เรียกได้ว่าเกือบจะไม่ต้องลงมือเขียนภาษา C จะคอมพายล์ออกมาเพื่อต้องการจะได้โปรแกรมทำงานสักอย่างหนึ่ง

ls ของยูนิกซ์

ไม่ต้องอะไรมาก ยกตัวอย่างเช่น /bin/ls ตัวเก่ง ซึ่งทำหน้าที่แสดงชื่อไฟล์คล้าย ๆ กับ dir ในดอส ในดอสนั้นทำอะไรได้บ้าง dir, dir/w, dir/p เพื่อดูชื่อแบบปกติ หรือแสดงเฉพาะชื่อตามแนวขวาง หรือแสดงทีละหน้าจอ และในรุ่นหลัง ๆ ก็มี /o เพื่อเรียงในลักษณะต่าง ๆ /a เพื่อเลือกดูตาม attribute ของไฟล์, /b เพื่อเอาแต่ชื่อเท่านั้น ไม่เอารายละเอียดอย่างอื่น, /l เพื่อให้แสดงเป็น lowercase หรือ /s เพื่อดูแบบ recursive ลงไปในไดเรคทอรีย่อยด้วย

แต่ /bin/ls ในยูนิกซ์ มีออปชันที่เหมือนกับ dir หมดยกเว้น /p ที่แสดงทีละหน้าเท่านั้นที่จะใช้โปรแกรมอื่นเข้าช่วย กล่าวคือปกติแล้วยูนิกซ์สายพันธุ์ system V ถ้าเราไม่ใส่ออปชันอะไรให้ มันจะแสดงเหมือน dir/b คือจะแสดงเฉพาะชื่อของไฟล์ชื่อละบรรทัด ส่วนถ้าเป็นสายพันธุ์ bsd จะคล้าย ๆ กับ dir/w คือเรียงตามแนวขวางให้แต่ว่าจะไม่มีบอกสรุปหรือชื่อ volumn ของดิสค์อะไรให้ ถ้าจะให้ ls ของ system V แสดงแบบขวาง ก็ใส่ออปชัน −C เข้าไป หรือถ้าจะให้ ls ของ bsd แสดงแต่ชื่อบรรทัดละชื่อก็ใส่ออปชัน −1 เข้าไป

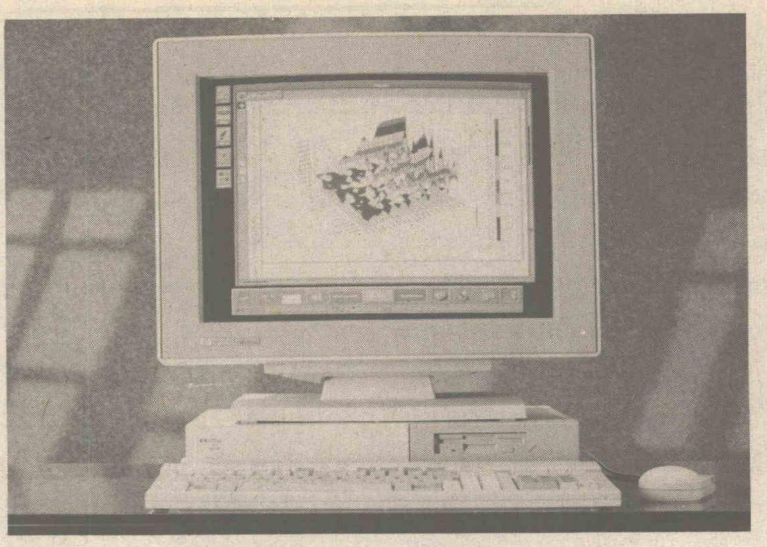
โดยปกติแล้ว ls จะแสดงชื่อให้เราดูโดยจะเรียงตามตัวอักษรของชื่อ เช่น ไฟล์ boy จะมาก่อนไฟล์ girl ('b' มาก่อน 'g') หรือ bang มาก่อน bomb เป็นต้น แต่ถ้าเราอยากให้เรียงตามเวลาก็ได้ โดยใส่ออปชัน −t เข้าไป ls ก็จะเรียงให้เราดูตามเวลาที่ไฟล์ถูกแก้ไขครั้งสุดท้าย โดยเรียงจากเวลาน้อยไปมากคือใหม่ไปเก่านั่นเอง

หรือถ้าอยากให้เรียงกลับ เช่นจาก 'z' มา 'a' หรือว่าจากเก่าไปใหม่ ก็เติมออปชัน −r เข้าไป นอกจากนั้นยังสามารถดูเลข i-node หรือคล้าย ๆ กับตำแหน่งของไฟล์บนดิสค์ ก็ใส่ออปชัน −i เข้าไป

หรือถ้าอยากดูขนาดเป็นจำนวน block ของไฟล์ (จำนวน block ของไฟล์บนยูนิกซ์นี้ก็คล้าย ๆ กับที่ในดอสเรียกว่าจำนวน sector) ก็ดูได้โดยเติมออปชัน −s เข้าไป

ส่วนการ recursive แบบ /s ของดอสนั้น ดูเหมือนจะเอามาจาก −R ของ ls บนยูนิกซ์ ทำงานคล้ายกันเลย และ /a ของดอสก็จะคล้าย ๆ กับ −a ของ ls แต่ว่าบนยูนิกซ์นี้ชื่อของไฟล์สามารถแอบซ่อนไฟล์ได้ คือ ls ปกติจะไม่แสดงไฟล์ที่ขึ้นต้นด้วยจุด '.' เช่น ไฟล์ที่สำคัญอย่าง .profile, .login หรือ .cshrc เป็นต้น ถ้าเราไม่ใส่ออปชัน −a เข้าไป ทั้งนี้เพื่อวัตถุประสงค์บางอย่างนอกเหนือจากการแอบคือ ความปลอดภัย ในกรณีที่เราเผลอไปเรียก rm * ซึ่งจะทำการลบไฟล์ทุกไฟล์ออก (อย่าง del *.* นั่นเอง) ไฟล์สำคัญอย่างพวก .profile นี้จะได้ไม่ถูกลบได้ง่าย ๆ วิธีหนึ่ง และอีกอย่างอาจจะเพื่อความสบายตาเวลา ls จะได้ไม่มองเห็นอะไรรกจนเกินไป

นอกจากนี้ยังมีออปชันอีกมากมายสำหรับ ls ซึ่งแต่ละเครื่องอาจจะแตกต่างกันบ้าง แต่นับว่ามีประโยชน์ทีเดียว จนดอสได้หยิบยืมความคิดไปเพิ่มความสามารถให้แก่ dir บ้างพอสมควร



ส่งผลลัพธ์ให้กัน

เพียงแค่ความสามารถของคำสั่งเดียวโดด ๆ อย่าง ls นี้ยังไม่พอ เชลล์ยังส่งเสริมให้มีการนำผลลัพธ์จากคำสั่งหนึ่งไปส่งต่อให้เป็น input ของคำสั่งอื่นได้ และที่จริงนี่แหละเป็นหลักการที่สำคัญอันหนึ่งเลยสำหรับระบบปฏิบัติการยูนิกซ์นี้ คือจะพยายามให้แต่ละโปรแกรมทำงานอย่างเดียว ไม่ใช่ทำได้ทุกอย่างที่ต้องการ เพื่อให้เกิดความง่ายของแต่ละส่วนเล็ก ๆ นั่นเอง แล้วถ้าอยากขยายขอบเขตก็ส่งผลลัพธ์ไปให้ตัวอื่นทำต่อ (ยกเว้นก็มีเหมือนกัน ก็อย่าง ls นี้ไง คงเนื่องจากเป็นคำสั่งที่ใช้บ่อย จึงได้ใส่เครื่องไม้เครื่องมือจิปาถะให้เต็มไปหมด)

การส่งต่อผลลัพธ์นี้ก็โดยอาศัย pipe หรือท่อซึ่งเป็นระบบ IPC (Inter Process Communication) อย่างง่าย ๆ ท่อนี้ช่วยให้เกิดประโยชน์ขึ้นมากมาย เช่นคำสั่ง 'ls | more' จะคล้ายกับ dir/p กล่าวคือ ผลลัพธ์จากคำสั่ง ls จะถูกส่งไปให้ more ซึ่งจะแสดงสิ่งที่เข้ามาให้มันออกไปทีละหน้า บนยูนิกซ์นี้ยังมี pg อีกตัวหนึ่งที่ทำงานคล้ายกัน แต่ว่า pg หรือ more ของยูนิกซ์นี้จะมีอะไรที่เหนือกว่า more บนดอสอยู่มากเลยเช่นสามารถพลิกดูหน้าเก่าได้ หรือสามารถหาข้อความที่ต้องการได้ด้วย โดยไม่ต้องใช้ตามองหาไปเรื่อย ๆ ทีละหน้า หรือจะ save สิ่งที่กำลังดูอยู่ลงเป็นไฟล์ด้วยก็ได้ ฯลฯ

ถ้าเราอยากให้ ls แสดงโดยเรียงตามขนาดของไฟล์โดยใช้ pipe ช่วยเรียก 'ls −l | sort +4 −n' คำสั่ง ls −l จะแสดงชื่อไฟล์พร้อมทั้งขนาดและอย่างอื่น ๆ อีก 3-4 อย่างไฟล์ละบรรทัดส่งไปให้โปรแกรม sort โปรแกรม sort จะนำข้อมูลที่ได้รับมานี้ไปเรียงบรรทัดโดยออปชันที่เราใส่ไปให้ คือ +4 จะให้ sort เรียงโดยใช้ฟิลด์ถัดไปอีก 4 คือฟิลด์ที่ 5 ซึ่งตรงกับฟิลด์ของขนาดไฟล์เป็นคีย์ในการเรียง และใส่ออปชัน −n เพื่อให้ sort มองข้อมูลในฟิลด์เป็นแบบตัวเลขไม่ใช่มองเป็นตัวหนังสือ

ลักษณะการใช้ pipe บนยูนิกซ์นี้เรียกได้ว่ามีประสิทธิภาพจริง ๆ แต่ละคำสั่งของยูนิกซ์จะส่งเสริมการใช้ pipe มากกว่าบนดอส ดูอย่าง dir หากเราจะ dir แล้วส่งต่อไปให้ตัวอื่นทำจะพบว่ามีหัวและหางอยู่พอสมควร คือส่วนที่บอก volume ดิสค์และขนาดพื้นที่ที่เหลือเป็นต้น อาจจะกระทบการทำงานของโปรแกรมที่รับข้อมูลไปได้เช่น sort จนกระทั่งดอสรุ่นหลัง ๆ มานี้ได้เพิ่มออปชัน /b เข้าไปตามที่ได้กล่าวมาแล้ว

เอาละ ในตอนนี้ อยากจะยกตัวอย่างการใช้เชลล์และโปรแกรมต่าง ๆ ที่ระบบยูนิกซ์มีไว้ให้เรา โดยเริ่มต้นจากของเล่นเล็ก ๆ น้อย ๆ อย่างนาฬิกา, โปรแกรมล็อกเทอร์มินัล และวิธีเก็บไดเรคทอรีเก่า ๆ แล้วเรียกกลับมาใช้ เพื่อแสดงถึงแนวคิดบางอย่าง และวิธีเบื้องต้นวิธีหนึ่งในการใช้ประโยชน์และความสามารถที่ยูนิกซ์มีไว้ให้

นาฬิกา

ผู้ที่ใช้คอมพิวเตอร์บางคนชอบที่จะให้มีนาฬิกาเดินอยู่บนหน้าจอ (ด้วยสาเหตุใดก็แล้วแต่) จึงทำให้มีโปรแกรมประเภทนี้มากมายบนดอส ตั้งแต่โปรแกรมเรสซิเดนต์แบบตัวเลขนับร้อย ๆ ที่เป็นพื้นฐานให้นักหัดเขียน Interrupt Service Routine (ISR) กับเรสซิเดนต์ไปจนถึงนาฬิกาเข็มเรือนสวยบนวินโดวส์ (WINDOWS)

ของเล่นเหล่านี้ ใครว่าบนยูนิกซ์ไม่มี? ที่จริงแล้วถ้าไม่นับนาฬิกาสำเร็จรูปที่แถมมาให้กับยูนิกซ์บางระบบ เช่น นาฬิกาบนระบบ openwin ของเครื่อง SUN เป็นต้น เราเองก็อาจจะทำนาฬิกาตัวเลขแบบง่าย ๆ บนยูนิกซ์ ซึ่งสามารถเขียนได้รวดเร็วกว่าบนดอสมาก ไม่ต้องเรสซิเดนต์จัดหน่วยความจำอะไร ไม่ต้องดักอินเตอร์รัพต์นาฬิกาให้วุ่นวาย และขนาดไม่ต้องใช้คอมพายเลอร์หรือแอสเซมเบลอร์เลยด้วยซ้ำ เพียงแค่มีเอดิเตอร์ที่คุณถนัดก็เพียงพอแล้ว (ที่จริงแล้วอาจจะเขียนสด ๆ บนเชลล์เลยก็ยังได้ แต่คงไม่มีใครที่ไหนทำกันหรอก เพราะมันคล้าย ๆ กับเราเขียนโปรแกรมภาษาแอสเซมบลีบนดอสโดยเขียนลงบนโปรแกรมดีบักอะไรทำนองนั้น)

เพราะความสามารถในการทำมัลติทาสกิง multitasking ของยูนิกซ์นั่นเอง เราเลยไม่ต้องคิดถึงการเรสซิเดนต์หรือจัดหน่วยความจำให้วุ่นวายและเพราะเชลล์ช่วยให้รันโปรแกรมแบบฉากหลัง (background) ได้คือจะรันโปรแกรมที่เราเรียกไปพร้อม ๆ กันกับการกลับมาให้เราควบคุมเชลล์ต่อไปได้เลยโดยไม่ต้องดักอินเตอร์รัพต์อะไรเอง ให้ระบบปฏิบัติการจัดการให้ และเพราะความคล่องตัวของภาษาเชลล์ และความมากมายของโปรแกรมที่ระบบจัดไว้ให้เราใช้ ใน /bin, /usr/bin ฯลฯ นั่นเองเลยไม่ต้องเขียนด้วย C หรือแอสเซมบลีอะไรเลย แค่เขียนโปรแกรมภาษาเชลล์หรือ 'เชลล์สคริปต์' (shell script) ก็ทำนาฬิกาอย่างง่าย ๆ ได้ (ต้องบอกว่าอย่างง่าย ๆ นะไม่ได้สวยหรูอะไร)

โปรแกรมในรูปที่ 1 เป็นเชลล์สคริปต์ของนาฬิกาตัวเลขแบบง่าย ๆ โดยใช้ bourne shell เนื่องจาก ปัจจุบันนี้เชลล์มีหลายตระกูลแล้วแต่ที่เรียกได้ว่ายูนิกซ์ทุกตัวจะต้องมีก็คือ bourne shell ซึ่งเกิดมาพร้อม ๆ กับยูนิกซ์ บรรทัดแรกจึงใส่ #!/bin/sh เพื่อเผื่อให้เรียกบนเชลล์ชนิดอื่นแล้วจะได้รันได้ (โดยเชลล์นั้นจะเรียก bourne shell ขึ้นมารันสคริปต์นี้)

บรรทัดที่ 9-13 จะคำนวณช่วงเวลาที่จะให้นาฬิกาแสดงเวลาทุก ๆ กี่วินาที จากบรรทัดที่ 9 นิพจน์ ${1-1} หมายความว่า ถ้าเรียกโปรแกรมโดยใส่พารามิเตอร์ก็ให้ใช้ค่านั้น คือ ใช้พารามิเตอร์ตัวที่ 1 แต่ถ้าไม่ใส่ ก็ให้ใช้ค่าเป็น 10 วินาที ที่ทำการ +0 ก็เพื่อจะตรวจสอบว่าพารามิเตอร์ที่ใส่มานั้นเป็นตัวเลขจริงหรือเปล่า เพราะถ้าไม่จริง expr จะให้ return code ไม่เป็นศูนย์ (ปกติโปรแกรมบนยูนิกซ์มักจะให้ return code เป็นศูนย์ถ้ารันได้สำเร็จ) แล้วเราจะตรวจสอบ return code ได้ด้วย $? ในบรรทัดที่ 10

บรรทัดที่ 14 ก็คือ ถ้าใส่พารามิเตอร์ให้ 2 เวลาเรียกโปรแกรมจะนำมาใช้เป็นสีสันในการแสดง (attribute) ซึ่งในจอเทอร์มินัลยูนิกซ์จำนวนไม่น้อยจะเป็น vt100 (หรือ vt100 เสมือน) ซึ่งจะใช้ escape sequence ตามที่โปรแกรมนี้ใช้ มีข้อแม้อีกสำหรับโปรแกรมนี้คืออาจจะต้องแก้ไขบ้างถ้าไปรันบนจอบางชนิด อาจจะต้องแก้ที่ attrb นี้รวมทั้งบรรทัดที่ 26 โดยดูได้จากคู่มือของจอชนิดนั้น ๆ ในส่วนของ escape sequence

ส่วนในบรรทัดที่ 15-20 ใช้คำสั่ง case เพื่อตรวจสอบค่าของ attrib อีกทีหนึ่งว่าเป็น BRI (bright) หรือ INV (inverse) หรือไม่ ถ้าใช่ก็เปลี่ยนเป็นส่วนหนึ่งของ escape sequence ที่ควบคุมสีสันในการแสดงผล ถ้าไม่ใช่ก็ฟ้อง error message ออกมา

ในบรรทัดที่ 21 ถึงสุดท้าย เห็นได้ว่ามีวงเล็บคร่อมไว้ และเติมเครื่องหมาย & ไว้ข้างหลัง เชลล์ที่รันอยู่นี้ก็จะรันส่วนที่อยู่ภายในวงเล็บเป็นแบบโปรเซสแบบฉากหลัง ซึ่งบรรทัดที่ 24 ถึง 28 จะวนลูปแบบนิรันด์ (forever loop) โดยทำการแสดงค่าจะคำสั่ง date ซึ่งผ่านการกรองด้วย cut พารามิเตอร์

```
1:  #!/bin/sh
2:  #-------------------------------------------------------------------------------
3:  # script          clock
4:  # version         1.0 (16 Sep 1991)
5:  #-------------------------------------------------------------------------------
6:  usage="usage: $0 [echo-time (sec)] [attrib (BRI, INV)]"
7:  #
8:  tt=`tty`
9:  etime=`expr ${1-10} + 0 2> /dev/null`
10: if [$? != 0]
11: then
12: etime=1
13: fi
14: attrib=${2-INV}
15: case $attrib in
16:       BRI) attrib=1 ;;
17:       INV) attrib=7 ;;
18:       *) echo $usage > /dev/tty
19:          exit 1
20: esac
21: (
22: time=`date | cut -c19`
23: sleep `expr 10 - $ time`
24: while true
25: do
26:       echo "^[7^[[1;73H^[[$attrib;m`date | cut -c12-19`^[[0m^[[8\c" > $tt
27:       sleep $etime
28: done
29: ) &
```

รูปที่ 1 เชลล์สคริปต์ของโปรแกรมนาฬิกาอย่างง่าย ๆ

ของ cut คือ −c12−19 หมายถึงให้แสดงคอลัมน์ที่ 12-19 จากสิ่งที่เป็น standard input ของมันออก standard output ซึ่งคำสั่ง echo จะรับไปแสดงออกทางจอที่สคริปต์นี้ถูกเรียก (จากคำสั่งในบรรทัดที่ 8)

ล็อกจอทำงาน

จะพบว่ามีอยู่หลายครั้งเหมือนกันเวลาที่เรากำลังใช้จอเทอร์มินัลอยู่แล้วอยากเดินไปที่อื่นเพื่อไปทำธุระอะไรบางอย่างนิดหน่อย แล้วกลับมาทำงานต่อ ถ้าหากเราปล่อยจอทิ้งไว้ภายใน login ของเรา อาจจะมีใครมือดีมาแกล้งอะไรเราได้ เช่นลบไฟล์บางไฟล์ทิ้ง หรือแอบดูข้อมูลลับบางอย่างของเราก็ได้ อันนี้เป็นเรื่องของระบบรักษาความปลอดภัยในการใช้เครื่องเบื้องต้นที่เราต้องระวัง โดยเฉพาะอย่างยิ่งหากเราใช้ login root อยู่ซึ่งอาจจะทำให้เกิดความเสียหายแก่ระบบทั้งระบบได้ เนื่องจาก login root นี้ทำได้ทุกอย่างไม่ว่าจะไปลบไฟล์, ดู หรือแก้ไขข้อมูลของยูสเซอร์คนไหนก็ได้ จึงต้องระวังเป็นพิเศษ

วิธีแก้ง่าย ๆ อย่างหนึ่งก็คือ logout ออกทุกครั้งที่จะเดินออกจากจอ แต่ถ้าเกิดไม่อยากเสียเวลามา logout และ login หลายครั้ง หรือถ้าในห้องนั้นมีจอน้อย ชนิดว่าต้องแย่งกันใช้แบบเล่นเกมเก้าอี้ดนตรีแล้วก็ การล็อกจอภาพด้วยโปรแกรมง่าย ๆ ก็เป็นทางเลือกหนึ่งที่ไม่เลวนัก เพียงแต่เรียก โปรแกรมก็จะถามรหัสเรา เพื่อใช้ตรวจสอบเวลาจะปลดการล็อก โดยจะถามสองครั้งแรกเพื่อตรวจสอบความแน่นอน ดูได้จาก listing ในรูปที่ 2 บรรทัดที่ 8-11 แล้วเทียบสองค่านี้ว่าตรงกันหรือไม่ในบรรทัดที่ 12 จะเห็นว่าการเทียบกันแบบไม่เท่ากันจะเขียนได้สองแบบคือ ใช้ != ติดกันอย่างในรูปที่ 1 บรรทัดที่ 10 หรือแยก ! กับ = กันก็ได้

```
 1: #!/bin/sh
 2: #-------------------------------------------------------------------------------
 3: # script          lock terminal
 4: # version         1.0 (17 Sep 1991)
 5: #-------------------------------------------------------------------------------
 6: MAXTRY=10
 7: stty -echo
 8: echo "\nEnter password: \c"
 9: read pwd1 < /dev/tty
10: echo "\nReenter again: \c"
11: read pwd2 < /dev/tty
12: if [! "$pwd1" = "$pwd2"]
13: then
14:     echo "\nîGIncorrect password. NO LOCKING."
15: else
16:     trap "" 2 3
17:     delay=1
18:     while true
19:     do
20:         echo "î[[2Jî[[Hî[[1mî[[5m\n\n\n"
21:         echo "                              ** Terminal Locking ** î[[0m"
22:         echo "\n\n\nPassword: \c"
23:         read password < /dev/tty
24:         if ["$password" = "$pwd1"]
25:         then
26:             break
27:         fi
28:         sleep $delay
29:         delay=`expr $delay + 1`
30:         if [$delay -gt $MAXTRY]
31:         then
32:             kill -9 `ps -lp $$ | tail -1 | cut -cut -c19-23`
33:         fi
34:     done
35:     echo "\n"
36:     trap 2 3
37: fi
38: stty echo
```

รูปที่ 2 สคริปต์โปรแกรมล็อกเทอร์มินัลอย่างง่าย ๆ

ในบรรทัดที่ 7 ใส่ stty −echo เพื่อไม่ให้เชลล์แสดงตัวอักษรที่เรากดออกทางหน้าจอ โปรแกรม stty นี้มีประโยชน์อย่างมากในการควบคุมระบบ input-output ต่าง ๆ ส่วนในบรรทัดที่ 16 เป็นคำสั่ง trap ซึ่งใช้ดัก signal ที่อาจเกิดขึ้น (signal นี้ก็อาจมองว่าคล้ายกับอินเตอร์รัพต์บนดอส ซึ่งเราสามารถดักได้เหมือนกัน) ในที่นี้ดักการกด ctrl−C โดยดัก SIGINT และดัก SIGQUIT ด้วย

ในบรรทัดที่ 23-27 ซึ่งอยู่ใน infinity loop จะอ่านรหัสจากคีย์บอร์ดโดยตรง จากคำสั่ง read password

< /dev/tty นั่นเอง และตรวจสอบกับรหัสซึ่งอ่านไว้แล้วก่อนหน้านั้น ถ้าตรงกันก็จะหลุดจากลูปด้วยคำสั่ง break แล้วจะเรียก stty echo เพื่อให้การกดคีย์ใด ๆ มีการแสดงตัวอักษรนั้นออกหน้าจอด้วยเหมือนปกติ

ส่วนในบรรทัดที่ 29 จะเพิ่มค่าหน่วงเวลาทีละ 1 วินาที ถ้าเกินกว่า MAXTRY (ในที่นี้คือ 10) จะถอด process shell ที่เรียกสคริปต์นี้ออกไป โดยการ kill process ในบรรทัดที่ 32

ที่จริงแล้วในระบบยูนิกซ์สมัยนี้มักจะมีโปรแกรมลักษณะนี้อยู่แล้ว แต่ในบางเครื่องที่ไม่มีสคริปต์สั้น ๆ นี้ก็จะเป็นประโยชน์อยู่บ้างไม่มากก็น้อย

เก็บไดเรคทอรีทำงาน

ในขณะที่เราทำงานบนเชลล์นั้น เนื่องจากโครงสร้างไดเรคทอรีของยูนิกซ์นั้นค่อนข้างใหญ่โต เช่นตำแหน่ง login ของเราอาจจะเป็น /u0/user/grad/35/johan เป็นต้น และยังมักจัดแบ่งไฟล์ต่าง ๆ ลงในไดเรคทอรีย่อยอีก จนบ่อยครั้งการย้ายไดเรคทอรีระหว่าง 2-3 จุด ทำให้เราต้องเขียนคำสั่ง cd ยาวมากลำบากและผิดพลาดได้ง่าย ในเชลล์สมัยใหม่ ๆ อย่าง korn shell มีความสามารถในการจดจำตำแหน่งไดเรคทอรีเก่าได้ เช่นคำสั่ง "cd –" จะย้ายเราไปไดเรคทอรีก่อนที่เราจะเรียก cd ครั้งหลังสุดเป็นต้น

แล้วถ้าเป็นเชลล์เก่า ๆ หละ? วิธีหนึ่งก็คือเก็บตำแหน่งไดเรคทอรีต่าง ๆ ไว้ในตัวแปรเชลล์ต่าง ๆ กัน เช่น WORK_P="$HOME/project/utility/working" export WORK_P เป็นต้น อย่างนี้เวลาเราจะย้ายไปที่ไหนเราก็เรียกสั้นลงอย่างเช่น cd $WORK_P ก็ได้

```
 1: #-------------------------------------------------------------------------------
 2: #
 3: # script            push/pop directory
 4: # version           0.1 (15 May 1991)
 5: # installation:
 6: #     append this script into your $HOME/.profile
 7: #
 8: #-------------------------------------------------------------------------------
 9: PPFILE=$HOME/.pushpop
10: TMPFILE=/tmp/.$LOGNAME$$
11: push( )
12: {
13:     if [$# -gt 0]
14:     then
15:         for arg
16:         do
17:             if [$arg = "."]
18:             then
19:                 pwd >> $PPFILE
20:             else
21:                 echo $arg >> $PPFILE
22:             fi
23:         done
24:     else
25:         pwd >> $PPFILE
26:     fi
27:     cat $PPFILE
28: }
29: pop( )
30: {
31:     if [-f $PPFILE]
32:     then
33:         dest='tail -1 $PPFILE'
34:         echo $dest
35:         cd $dest
36:         lines='cat $PPFILE | wc -1'
37:         lines='expr $lines - 1 | sed "s|[ ]* \([^ \n\)*|\1|"'
38:         if [$lines -gt 0]
39:         then
40:             sed -n "1,$lines p" $PPFILE > $TMPFILE
41:             mv $TMPFILE $PPFILE
42:         fi
43:     else
44:         echo Stack of directory is empty
45:     fi
46: }
```

รูปที่ 3 สคริปต์ฟังก์ชัน push/pop ไดเรคทอรีที่ควรเก็บไว้ใน .profile

แล้วถ้าเราจะต้องทำงานกับหลาย ๆ ไดเรคทอรีแล้วก็จะต้องตั้งชื่อตัวแปรเชลล์ให้ต่าง ๆ กันไป มีอีกวิธีหนึ่งคือใช้เก็บลงในไฟล์ ซึ่งจะทำให้ทำงานช้ากว่าแต่ว่าเก็บได้มากกว่าและในการเก็บนี้ก็จะเก็บในลักษณะของสแต็ก (stack) คือเข้าก่อนออกทีหลัง

ในรูปที่ 3 เป็นสคริปต์ที่จะใส่ไว้ใน

.profile ซึ่งจะถูกเรียกทุกครั้งที่ login เข้ามาเพราะในการที่จะเปลี่ยนไดเรคทอรีที่กำลังทำงานอยู่นั้นจะต้องทำโดยเรียกใช้ฟังก์ชันหรือเรียกแบบไม่ให้สร้างเชลล์ใหม่ขึ้นมา คือ เรียกแบบมี '.' ทำหน้าสคริปต์ (ใน bourne shell) หรือใช้คำว่า source นำหน้า (ใน C shell) ซึ่งจะทำให้เรียกลำบาก แต่การใช้ฟังก์ชันลักษณะนี้มีข้อเสียอยู่ตรงที่ว่า ถ้าเราเรียกเชลล์อย่อยขึ้นมาโดยไม่ไปผ่าน .profile จะทำให้เชลล์ใหม่นั้นไม่รู้จักฟังก์ชันนี้

ฟังก์ชันที่ใช้ก็มีอยู่ 2 ตัวคือ push กับ pop ฟังก์ชัน push จะทำหน้าที่บันทึกไดเรคทอรีที่เราต้องการลงในไฟล์ ถ้าเรียกโดยไม่ใส่อาร์กิวเมนต์ใด ๆ ก็จะเอา working directory หรือไดเรคทอรีที่กำลังอยู่ขณะนั้นเก็บไว้ ส่วนฟังก์ชัน pop จะทำหน้าที่เรียกไดเรคทอรีล่าสุดที่เราเก็บไว้ออกมา

ในบรรทัดที่ 15-23 จะเห็น for arg ... do ... done คำสั่ง for ตามด้วยชื่อตัวแปรเชลล์นี้ หมายถึงจะให้ตัวแปรเชลล์นั้นแทน positional parameter ทีละตัว เช่นถ้าเราเรียก push/$HOME คำสั่งภายใน do ถึง done ก็จะทำ 2 ครั้ง โดยค่าใน arg ก็จะเป็น / และค่าที่เก็บใน HOME (ซึ่งปกติจะเป็นไดเรคทอรีเวลา login นั่นเอง) ตามลำดับ

ส่วนในบรรทัดที่ 37 และ 40 จะเห็นการใช้โปรแกรม sed (stream editor) ซึ่งจัดได้ว่าเป็นโปรแกรมที่มีประสิทธิภาพสูงมาก (ถ้ารู้จักใช้) เป็นโปรแกรมบนยูนิกซ์ตัวหนึ่งที่น่าเรียนรู้ไว้ถึงแม้จะใช้ค่อนข้างยาก

รูปที่ 4 สรุปการใช้คำสั่งของ bourne shell

SHELL Standard Command Language

sh [-ceiknrstuvx] [*arg* ...]
execute commands
-c *string*
commands are read from *string*
-e if non-interactive, exit on bad status
-i interactive shell
-k all keyword arguments are placed in the environment for a command
-n read commands but do not execute them
-r call a restricted shell
-s commands are read from standard input
-t exit after reading and executing one command
-u treat unset variables as error when substituting
-v print shell input lines as they are read
-x print on execution
- no handling of -v and -x
-- set $1 to -

SHELL Command Interpreter

Special Shell Characters

Character	Meaning
|	Pipe
;	Sequential command separator
&	The shell does not wait for preceding command to finish (asynchronous execution).
&&	The following command is to be executed only if the preceding command returns a zero exit status.
||	The following command is to be executed only if the preceding command returns a non-zero exit status.
'...'	All characters enclosed between a pair of single quote marks are taken literally. Special characters are quoted.
"..."	Inside double quote marks, parameter and command substitution occurs.
(...)	Embrace several commands.
\	Nullify special meaning of the following character.
`...`	The standard output from a command enclosed in a pair of grave accents may be used as part or all of a word

Parameter Substitution

Positional parameters (name is a digit): values are assigned by set or by submission on invoking a procedure.
Variables or keyword parameters (name begins with a character): values are assigned by writing *name=value*

${*parameter*}
The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name.
${*parameter*[:]-*word*}
If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*
${*parameter*[:]=*word*}
If *parameter* is not set or is null set it to *word*; the value of the parameter is substituted
${*parameter*[:]?[*word*]}
If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, message "parameter null or not set" is printed
${*parameter*[:]+*word*}
If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

If the colon (:) is omitted from the above expression, the shell only checks whether *parameter* is set or not.

Standard Variables

Name	Contents
#	Number of positional parameters in decimal
?	Decimal value returned by the last synchronously executed command
$	Process number of this shell
!	Process number of the last background command
-	Flags supplied to the shell on invocation or by set
*	All parameters of a procedure
0	Argument 0 on invoking a shell
HOME	Default argument for cd
PATH	Search path for commands
MAIL	File for mail-messages
PS1	Primary prompt string, by default $
PS2	Secondary prompt string, by default >
IFS	Internal field separators, normally space, tab, and new-line
SHELL	Pathname of the shell interpreter
TERM	Terminal type

รูปที่ 4 (ต่อ)

File Name Generation (Pattern Matching)

Character	Meaning
?	Matches any single character
*	Matches any string, including the null string
[...]	Matches any of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive.
[!...]	Matches any character not enclosed.

Input/Output Redirection

Character	Meaning
< *file*	Standard input is *file*
> *file*	Standard output is *file*. If *file* exists already, it is truncated to zero length.
>> *file*	Same as > but append to *file* if it exists
<<[-] *word*	Shell input is read up to a line that is the same as *word*. If any character of *word* is quoted (\), no interpretation is placed upon the characters; otherwise, parameter and command substitution occurs. If – is appended to <<, all leading tabs are stripped.
<&*digit*	Standard input is the file with the file descriptor *digit*.
>&*digit*	Standard output is the file with the file descriptor *digit*.
<&-	Close standard input.
>&-	Close standard output.

Special Commands

The following commands are executed by the shell interpreter without creating a new process:

break	case	cd	continue	eval	exec	exit
export	for	if	newgrp	read	readonly	set
shift	test	times	trap	ulimit	umask	until
wait	while		:			

Shell Command Language

if *command list*
then *command list*
fi
Conditional execution

if *command list*
then *command list*
else *command list*
fi
Simple control transfer

if *command list*
then *command list*
elif *command list*
then *command list*
...
[else *command list*]
fi
Nested simple control transfer

case *word* in
pattern1) *command list*;;
pattern2) *command list*;;
...
esac
Multiple control transfer

for *name* [in *word* ...]
do *command list*
done
Loop is executed a specified number of times

while *command list*
do *command list*
done
Loop is executed repeatedly until the last command in the *list* following while supplies a return code other than 0

until *command list*
do *command list*
done
Loop is ecexuted repeatedly until the last command in the *command list* following until supplies a return code of 0.

(*command list*)
Execute *command list* in a sub-shell.

{*command list*;}
Syntactical grouping of the specified commands.

บทส่งท้าย

ในรูปที่ 4 เป็นสรุปย่อ ๆ เกี่ยวกับเชลล์โปรแกรมมิง คิดว่าพอจะมีประโยชน์อยู่บ้างไม่มากก็น้อย และที่เล่ามาทั้งหมดนี้ จะเป็นการนำเสนอวิธีใช้งานเชลล์อย่างง่าย ๆ สคริปที่ให้มานี้หลาย ๆ คำสั่งอาจจะเขียนให้ดีกว่านี้ก็ได้ เพิ่มเติมความสามารถเข้าไป หรือแก้อะไรที่เกิน ๆ หรือยุ่ง ๆ ออกไปได้ อย่างในรูปที่ 3 ฟังก์ชัน pop อาจจะไม่ต้องใช้ sed ก็ได้ ที่เลือกใช้ก็เพียงเพื่ออยากแนะนำให้รู้จักโปรแกรมที่มีประโยชน์ตัวนี้เท่านั้น สำหรับรายละเอียดของสคริปต์ทั้งหลายเหล่านี้อยากจะให้อ่านดูควบคู่กับตารางในรูปที่ 4 เพื่อเป็นแนวทางในการประยุกต์ใช้เชลล์สคริปต์ต่อไป บางโอกาสจะได้ไม่ต้องเขียน C แล้วคอมพายล์อะไรให้ลำบาก